DIS Lab

Christian Krieg and Severin Jäger

Nov 28, 2024

CONTENTS:

CHAPTER

ONE

OVERVIEW

Welcome to this FPGA lab. The following instructions are to be carried out on the `iceBreaker` board, which you picked up earlier this year. You should have completed Task 0 already, which means your setup is working and you are ready to go!

Rough Timeline

08:00	13:00	Welcome, introduction to Task 1
08:15	13:15	Solve Task 1
09:30	14:30	Discussing the results of Task 1
10:00	15:00	Introduction to Task 2 (including HTerm)
10:10	15:10	Solve Task 2
11:30	16:30	Discussing the results of Task 2
11:50	16:50	Wrap-Up & Feedback

CHAPTER

TWO

TASK 1 - BRING YOUR COUNTER TO HARDWARE

Note:

- Time for solving the task: 2 hours
- Points for successful solving: 26

So let's start. In the preceding VELS tasks, we transformed an idea into a specification; we implemented the specification and synthesized it into a bitstream. We analyzed various synthesis paths in order to find an optimal implementation of our counter. We identified Pareto-optimal implementations.

Your task is now to take the Pareto-efficient implementation of your choice, and to bring it on your board. During this process, we will investigate the following steps:

1. Write a physical constraint file (PCF) to connect your design to buttons and LEDs
2. Analyze the resource consumption of your design
3. Program the FPGA
4. Automate all steps with a make file to apply changes quickly

For this task, you need the following files from the previous exercises:

- `counter.v` (behavioural RTL version)
- `map.v`
- `tb_counter.sv`
- `counter.sby`

To simplify automation we provide you the following files. Both of them are not complete yet, you will complete them during the task.

- `synth.ys`
- `Makefile`

There will not be any submission system, but be prepared for a discussion about your work and your results. We will check whether you understood the concepts we taught throughout the exercise.

2.1 Synthesis of the Pareto-efficient design

Firstly, complete `synth.ys` to synthesise one of the designs you found out to be Pareto optimal during the last task and to create the corresponding `.json` file. We will reuse the script later on to quickly modify our design.

2.2 Write a physical constraint file

Please be aware that we need to map some of the inputs and outputs to external pins, so we need to write a new constraints file (`.pcf`) in order to define these mappings. Use the board documentation to find the relevant pins. Use the following command for all relevant inputs and outputs:

```
1 set_io signal_name port_number
```

Please map your inputs `mode`, `reset`, `clk` to the buttons such that you can conveniently operate them. Please map at least four relevant bits of your output to the LEDs on the board (do not use the LEDs connected to pins 11 and 37, you might need them later on). This will enable us to check the correctness of the counter value.

2.3 Place & Route

Let us revisit what we did in task 7. We exported a `.json` file from `Yosys` and used `nextpnr` to place and route the counter for an FPGA. Please look at the details in the board documentation and use `nextpnr-ice40 -h` to figure avalilable options. Then place and route your counter design using the constraint file we created and create a `.asc` file.

Finally, look at the resource consumption and the timing information in the `nextpnr` output. Did it change relative to the table you created and if so, why? Which resource will become critical if we instantiate our counter multiple times?

2.4 Program the FPGA

Use `iceprog` to program your FPGA. Note that you have to pack you design first using `icepack`.

Play around with your counter design on the board. Check whether it works correctly (e.g. whether the reset is synchronous). If you think that you did not chose suitable constraints adjust your `.pcf` file so you can showcase your counter.

Important: Talk about bouncing: what can we do about it?

2.5 Automation

You probably did all your design flow steps by calling tools in the command line. If you make changes to your design (like we will do in the following) you want to be able to redo them quickly. One way to do so are bash scripts, but here we will use a make file to reproduce our work. With make files, we conveniently can organize automation tasks (we call them *targets*), and easily perform them just by using the `make` command.

Complete the provided `Makefile` by adding the commands from the previous subtasks so you can verify with `make check`, synthesize with `make synth`, and place and route your design with `make place`. A simple call to `make all` does all these tasks for you. Furthermore, call the target `prog` to flash your FPGA conveniently.

Important: Makefiles only accept tabs (no spaces for indentation)

2.6 Modify the design

Finally, we want to modify our design so we can benefit from our `make` tool chain. Your task is the following: Add two top-level signals to your design. One of them should indicate that you are currently close to the counting range (so the counter value does not change, which is near the minimum/maximum). Map it to the remaining red LED. The second signal should be on if you are currently skipping the forbidden counter value (i.e. when the next clock edge arrives). Use the green LED for demonstration. Use the make target `all` to verify that your new counter still does what you expect and to undergo the design flow. Finally, check your changes on the FPGA.

Make sure your design still only uses one single adder. Again, take a look at the resource consumption and the maximum clock frequency of your design. Did the achievable clock frequency change? Why? Or, why not?

Note: Finalize your Makefile!

Important: **Verification challenge:** `tb_counter.sv` uses `.*` binding, which is not applicable to the modified outputs

CHAPTER

THREE

TASK 2: INTEGRATE YOUR COUNTER INTO A HIGHER-LEVEL APPLICATION

Note:

- Time for solving the task: 2 hours
- Points for successful solving: 25

Now that we brought our counter on hardware, and did some initial testing (with our fingers providing input and with our eyes verifying its output), it is now time to integrate our counter into a higher-level application. We assume a host machine that outsources counting to an external accelerator (i.e., your counter). Of course, we cannot manually verify our counter anymore, which is why we use a serial interface (UART) to communicate with our counter. This requires us to do the following steps:

1. Instantiate a UART interface in our design (We provide that interface)
2. Connect the UART to the counter
3. Connect to the glue logic between the UART and the counter to provide input (`clk`, `mode`, `rst`) and access the output (`cnt`). Watch out for **TODO** notes in the file `counter_uart_top.v`.
4. Set up a serial interface on the host machine using `HTerm`. From the terminal, we can find the serial port with the following command:

```
sudo dmesg | grep tty
```

5. Send some test data with `HTerm` to the counter from the host, do the counting on the accelerator, and send the counter value to the host
6. In parallel, calculate the expected counter value, and compare it to the actual counter value

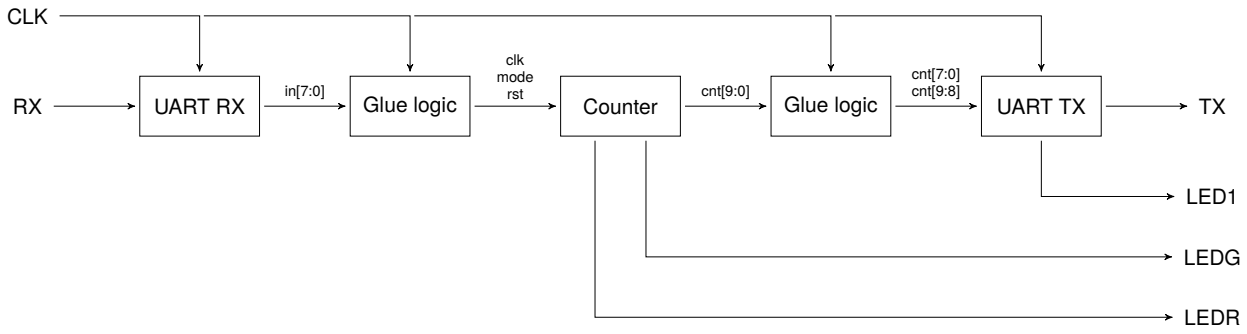
For this task, you need the following files from lab task 1:

- `counter_pareto.v`

We provide you the following files.

- `Makefile`
- `counter_uart_top.v`
- `uart_rx.v`
- `uart_tx.v`
- `uart_baud_tick_gen.v`
- `counter_uart.pcf`

The following block diagram illustrates the desired design:



Use `HTerm` to interface with your design!